A Case Study on Prototyping Power Management Protocols for Sensor Networks*

Mahesh Arumugam, Limin Wang, and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing MI 488824
{arumugam, wanglim1, sandeep}@cse.msu.edu

Abstract. Power management is an important problem in battery powered sensor networks as the sensors are required to operate for a long time (usually, several weeks to several months). One of the challenges in developing power management protocols for sensor networks is prototyping. Specifically, existing programming platforms for sensor networks (e.g., nesC/TinyOS) use an event-driven programming model and, hence, require the designers to be responsible for stack management, buffer management, flow control, etc. Therefore, the designers simplify prototyping their solutions either by implementing their own discrete event simulators or by modeling them in specialized simulators. To enable the designers to prototype power management protocols in target platform (e.g., nesC/TinyOS), in this paper, we use *ProSe*, a programming tool for sensor networks. ProSe enables the designers to specify their programs in simple abstract models while hiding low-level challenges of sensor networks and programming-level challenges. As a case study, in this paper, we specify a power management protocol with ProSe, automatically generate the corresponding nesC/TinyOS code, and evaluate its performance. Based on the performance results, we expect that ProSe enables the designers to rapidly prototype, quickly deploy, and easily evaluate their protocols.

1 Introduction

In the recent years, sensor networks have become popular due to their wide variety of applications including border patrolling, hazard detection, habitat monitoring, and micro-climate monitoring. These applications require the network to operate for a long time (usually, several weeks to several months). However, the sensors are typically battery powered (e.g., Mica [1], XSM [2], Telos [3]) and, hence, they can operate continuously only for a few days. In addition, since the sensors are deployed in large numbers and mostly in inaccessible fields, it is difficult to change the batteries after deployment. Therefore, power management is crucial for extending the lifetime of the network.

* This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF Equipment Grant EIA-0130724, and a grant from Michigan State University.

One of the challenges in designing power management protocols for sensor networks is prototyping. Specifically, existing platforms (e.g., nesC/TinyOS [4]) for programming sensor networks use *event-driven programming model* and, hence, require the designer be responsible for stack management, buffer management, and flow control [5, 6]. Therefore, to rapidly prototype and quickly evaluate protocols, the designers of existing power management protocols (e.g., [7, 8, 9, 10, 11, 12, 13]) implement their own simulators or model their protocols in specialized simulators (e.g., GloMoSim [14]). However, it is desirable that the designers prototype their protocols in nesC/TinyOS platform as it provides a framework for generating both simulation as well as production code from the same source.

In this paper, we consider the problem of rapid prototyping of power management protocols in nesC/TinyOS platform. To deal with programming level challenges (e.g., stack management, buffer management, flow control, etc) and network level challenges (e.g., message collision, corruption, synchronization, etc) of sensor networks, we focus on *ProSe* [15], a programming tool for rapid prototyping of sensor network protocols and applications. ProSe is based on the theoretical foundation on computational model in sensor networks [16, 17]. It enables the designers to (i) specify programs in simple abstract models (e.g., shared-memory model, read/write model) that hide several challenges of sensor networks, (ii) automatically transform the programs into a model consistent with sensor networks, and (iii) automatically generate and deploy (nesC/TinyOS) binary.

In addition, we note that the transformation algorithms proposed in [16, 17] preserve self-stabilization and fault-tolerance properties of the programs in shared-memory model or read/write model in the transformed programs. Since we implement the transformation algorithms proposed in [16,17] in ProSe, ProSe automates the process of transformation of abstract programs. And, it preserves the self-stabilization and fault-tolerance properties of the transformed programs. (We refer the reader to [16, 17, 15] for more details on preserving properties of original programs.)

As a case study, we model *pCover* [13], a power management protocol that provides partial (but high) sensor coverage of the target field, in ProSe. We specify the pCover program in shared-memory model. We synthesize the corresponding nesC/TinyOS binary and study the performance of the generated code. Through simulations, we show that the generated program extends the lifetime of the network while providing a partial (but high) coverage.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we briefly discuss how programs are specified in ProSe. Then, in Section 3, we prototype pCover in ProSe. We present a brief overview of the protocol and discuss how we synthesized the nesC/TinyOS binary. Subsequently, we study the performance of the generated binary code. We show that the generated program extends the lifetime of the network. In Section 4, we discuss the lessons learned in prototyping power management protocols and in Section 5, we discuss the related work. Finally, in Section 6, we make the concluding remarks.

2 ProSe: Overview

In this section, we briefly outline the structure of programs in ProSe and discuss how nesC/TinyOS binaries are synthesized.

2.1 Structure of Programs

In ProSe, programs are specified in terms of guarded commands [18]; each command (or action) is of the form:

$$guard \quad \longrightarrow \quad statement,$$

where *guard* is a predicate over program variables, and *statement* updates program variables. An action $g \longrightarrow st$ is enabled when g evaluates to true and to execute that action, st is executed. A computation of this program consists of a sequence $s_0, s_1, \ldots$, where s_{j+1} is obtained from s_j by executing actions in the program $(0 \leq j)$.

Computation model. A computation model limits the variables that an action can read and write. Towards this end, we split the program actions into a set of processes (sensors). Each action is associated with one of the processes (sensors) in the program. We now describe how we model the restrictions imposed by shared-memory model and read/write model.

Shared-memory model. In this model, in one atomic step, a sensor can read its state as well as the state of its neighbors and write its own (*public* and *private*) variables.
Read/Write model. In this model, in one atomic step, a sensor can either (1) read the state of one of its neighbors and update its *private* variables, or (2) write its own variables.

Programs written in shared-memory model or read/write model, however, are not suitable for the constraints (and opportunities) provided by sensor networks. For this reason, in [16,17], the authors have modeled the computations in sensor networks as a *write all with collision* (WAC) model, discussed next.
Write all with collision (WAC) model. In this model, each sensor consists of write actions (to be precise, write-all actions). Specifically, in one atomic action, a sensor can update its own state and the state of all its neighbors. However, if two or more sensors simultaneously try to update the state of a sensor, say k, then the state of k remains unchanged. Thus, this model captures the fact that a message sent by a sensor is broadcast. But, if multiple messages are sent to a sensor simultaneously then, due to collision, it receives none.

To simplify programming sensor networks, recently, approaches have been proposed for transforming programs into WAC model. They can be classified as: (a) TDMA based deterministic transformation [16] and (b) CSMA based probabilistic transformation [17]. With the help of these transformation algorithms, ProSe allows the designer to specify programs in simple abstract models (e.g., shared-memory model, read/write model). Then, ProSe automatically transforms them into WAC model and, subsequently, generates the corresponding nesC/TinyOS code.

2.2 Input/Output of ProSe

The input to ProSe consists of the guarded commands program in shared-memory or read/write model, its initial states and (optionally) the topology of the network. We discuss the input/output of ProSe in the context of an example.

Input guarded commands program. Consider a MAX program, where each process (i.e., sensor) maintains a public variable x. The goal of MAX is to eventually identify the maximum value of this variable across the network. We specify the actions of each process in this program as shown in Figure 1 (keywords are shown in bold font):

```
program max
sensor j
var public int x.j;
begin
   (x.k > x.j) -> x.j = x.k;
end
init state x.j = j;
```

Fig. 1. MAX program in ProSe

The designer also specifies zero or more initial states of the program. If no initial states are specified, ProSe initializes the variables of the program to arbitrary values. If more than one initial states are specified, ProSe initializes the program to randomly selected state. In the above program, $x.j$ is initialized to j (i.e., ID of the sensor).

Auxiliary variables. ProSe provides abstractions to deal with failure of sensors and presence of Byzantine sensors. To determine whether a neighbor (say, k) is alive or failed, sensor j can just access the public variable *up.k*; if *up.k* is $TRUE$ (respectively, *FALSE*) then k is alive (respectively, failed). Designers can use this abstract variable to simplify the design of sensor network protocols while ProSe provides implementation of this variable through heartbeat protocol (e.g., [19]). Similarly, ProSe also allows designers to model Byzantine sensors through abstract variables ($b.j$).

Topology information. ProSe *wires* a component (*NeighborState*) that maintains the state information of the neighbors at each sensor, with the generated code. Towards this end, each sensor should identify its neighborhood. ProSe allows the designers to integrate a neighborhood abstraction layer (e.g., [20]) with the generated code. Such an abstraction layer allows a sensor to learn its neighborhood dynamically. Optionally, the designers can specify the static topology of the network as an input to ProSe using the *topology file*. This file includes the ID of the base station, size of the network, and the communication topology. Based on the neighborhood information, ProSe configures the MAC layer and NeighborState component.

Support for *local* component invocations in guarded commands. Since ProSe allows the designers to specify programs in guarded commands format, it makes protocol design highly intuitive and concise. However, it is not always desirable to use guarded commands to specify protocols. For example, consider the design of a routing protocol for sensor networks, where the sensors maintain a spanning tree rooted at the base station. In this program, whenever the parent of a sensor fails, it chooses one of its active neighbors for which the link quality is greater than a certain threshold, as its parent. Towards this end, the sensor has to compute the link quality of each of its neighbors. Specifying this action in guarded commands is difficult. Moreover, nesC/TinyOS components may exist that provide the desired functionality.

To simplify the design of sensor network protocols, ProSe allows component invocations in guarded commands. In the design of routing protocol, in order to find a neighbor that has a better link quality, the designer can invoke the component *LinkEstimator* to compute the quality estimate of a given link. Thus, parent update action in the routing protocol can be specified in guarded commands as shown in Figure 2.

```
// current parent (p.j) has failed and j-k link quality is greater than the threshold
(up.(p.j) == FALSE) && (up.k == TRUE) && (LinkEstimator.getQuality(k) > LINK_THRESHOLD)
  -> p.j = k; currentParentLinkQuality.j = LinkEstimator.getQuality(k);
```

Fig. 2. Component invocation in ProSe

In the above action, the *getQuality(k)* method of LinkEstimator component returns the quality of the link $j-k$. This component may need certain variables to compute the quality estimate. For example, it may need counters that maintain the number of messages successfully transmitted over each link. The action by which the counters are updated would be specified in guarded commands. The variables used in the guarded commands program and the copies of the public variables of the neighbors (maintained in NeighborState) are made available to the invoked component.

The designer has to implement *LinkEstimator* in nesC/TinyOS platform. This component, however, uses only local data (i.e., it uses NeighborState). ProSe generates the code for NeighborState component. And, it wires the component implemented by the designer with the generated code.

Output nesC/TinyOS code. In the generated nesC/TinyOS program, the actions of the input program are executed whenever a timer fires. Once the sensor executes each action for which the corresponding guard is enabled, it marshals all the public variables as a message *wacMsg* and schedules it for transmission (broadcast). Depending on the transformation algorithm and the MAC layer selected by the user, it configures when the timer fires and how *wacMsg* is transmitted. For example, in case of a TDMA based transformation [16], ProSe configures the timer to fire in every TDMA slots assigned to the sensor. And, it uses the TDMA service (e.g., [16, 21, 22]) to broadcast the message. In case of a

CSMA based transformation [17], ProSe configures the timer to fire in a random interval whenever it receives a message containing values of public variables at the sender. And, it uses a CSMA service (e.g., [23]) to broadcast *wacMsg*.

Similarly, ProSe generates code for NeighborState component that maintains the state information of the neighbors whenever it receives an update message from one of its neighbors. Finally, ProSe also generates code to (1) initialize all the program variables, (2) configure network services (e.g., TDMA, CSMA), and (3) configure and start middleware services (e.g., Timer).

3 Case Study: Prototyping pCover with ProSe

In this section, we present a case study on prototyping power management protocols with ProSe. We model *pCover* [13], a simple power management protocol that provides partial (but high) sensor coverage of the target field. Specifically, pCover maintains a certain degree of coverage through sleep-awake scheduling of sensors. By trading little sensor coverage of the field, in [13], the authors show (using C++ discrete event simulator) that pCover substantially improves the network lifetime.

First, in Section 3.1, we discuss the pCover program (written in shared-memory model). Then, in Section 3.2, we show how we synthesize the corresponding nesC/TinyOS binary with ProSe. Finally, in Section 3.3, we evaluate the performance of the generated code.

3.1 pCover: Overview

The pCover program written in shared-memory model is shown in Figure 3. The basic idea of pCover is that a sensor should turn itself off if and only if its *local coverage* is higher than a certain threshold, called *OnThreshold*. Local coverage of a sensor is the percentage of the sensor's sensing area that is covered by other awake sensors.

Description of the program. In this program, each sensor is in one of 4 states: *probe*, *awake*, *readyoff*, and *sleep*. Each sensor j maintains one public variable *st.j* that identifies the state of the sensor. In addition, j maintains a copy of the public variables of its neighbors (in NeighborState). We discuss the actions of the pCover program shown in Figure 3 in detail, next.

Probe state. A sensor in probe state probes the environment, determines whether it should stay awake or go to sleep. After a timeout Y, the sensor computes its local coverage. Note that the designer has to provide the *LocalCoverage* component that returns the local coverage of a sensor. This component acts only on the state information of the neighbors maintained at the sensor. The sensor starts working if its local coverage is lower than the OnThreshold. Otherwise, the sensor switches to sleep state. The timeout Y is used to ensure that when the sensor decides whether it should stay awake or go to sleep, it has the *fresh* state information of its neighbors.

```
program pCover
sensor j
const int X, Y, Z, S, W, OnThreshold, OffThreshold;
var
   public int st.j;
   private int timer.j;
component LocalCoverage;
begin
(st.j == SLEEP) && (timer.j >= X)
   -> st.j = PROBE; timer.j = 0;
| (st.j == PROBE) && (timer.j >= Y) && (LocalCoverage.compute() > OnThreshold)
   -> st.j = SLEEP; timer.j = 0;
| (st.j == PROBE) && (timer.j >= Y) && (LocalCoverage.compute() <= OnThreshold)
   -> st.j = AWAKE; timer.j = Random(0, S);
| (st.j == AWAKE) && (timer.j >= Z)
   -> st.j = READYOFF; timer.j = 0;
| (st.j == READYOFF) && (timer.j >= W)
   -> st.j = AWAKE; timer.j = Random(0, S);
| (st.j == READYOFF) && (LocalCoverage.compute() > OffThreshold)
   -> st.j = SLEEP; timer.j = 0;
| ((st.j == SLEEP) && (timer.j <= X)) ||
  ((st.j == PROBE) && (timer.j <= Y)) ||
  ((st.j == AWAKE) && (timer.j <= Z)) ||
  ((st.j == READYOFF) && (timer.j <= W))
   -> timer.j = timer.j + 1;
end
```

Fig. 3. pCover program in ProSe

Awake state. A sensor in awake state actively monitors the area within its sensing range. It remains active until the timer reaches the timeout value Z. Since we do not want all awake sensors to timeout at the same time, the timer is initialized to a random value. Once the awake timer expires, the sensor changes its state to *readyoff*.

Readyoff state. In readyoff state, the sensor still provides sensing coverage. However, the neighbors of a readyoff sensor (say, j) consider j as a sleeping sensor. In other words, the neighbors of j do not count it when they compute local coverage. If a readyoff sensor finds that its local coverage is greater than *OffThreshold*, it will change its state to sleep. Also, if a sensor is in readyoff state for a long duration, it can switch to awake state. This action allows one to deal with the case where a lot of sensors are in readyoff state although none of them can go to sleep state (due to local coverage being less than OffThreshold).

Sleep state. A sensor in sleep state wakes up every X minutes. When it wakes up, it changes its state to probe and proceeds to execute actions in that state.

3.2 Transformation and Code Generation

We use ProSe to generate the nesC/TinyOS implementation of the pCover program and subsequently build the binary image. Towards this end, we use the TDMA based transformation from [16] to transform the program into WAC model and generate the nesC/TinyOS code. We integrate SS-TDMA [21] with the generated program to implement the write-all action. As mentioned in Section 2, since the pCover program includes component invocation (LocalCoverage) in the actions, we require the designer of the protocol to implement this component in nesC/TinyOS. We discuss how the designer implements this component and how ProSe integrates it with the generated code, next.

LocalCoverage component. Based on the state information of the neighbors of a sensor (say, j), LocalCoverage component computes the percentage of j's sensing area that is covered by its *awake* neighbors. This component provides a method (*compute()*) that could be invoked in the guarded commands program. This method returns the local coverage of the sensor.

In order to compute the local coverage of the sensor, LocalCoverage requires the state information of the neighbors of the sensor. This information is maintained by NeighborState component (as mentioned in Section 2). Since, ProSe wires NeighborState with LocalCoverage when generating the nesC/TinyOS code for pCover, LocalCoverage component can obtain the state information of the neighbors of the sensor by invoking NeighborState. Note that all accesses to NeighborState are local and ProSe is responsible for updating NeighborState with *fresh* values. Thus, the designer does not have to deal with programming level challenges of nesC/TinyOS platform and low-level challenges of sensor networks (e.g., communication, collisions, corruption, etc).

3.3 Evaluation of the Synthesized Program

We evaluate the performance of the generated nesC/TinyOS code for pCover with TOSSIM [24], a discrete event simulator for TinyOS sensor networks.

Simulation settings. We use the simulation setting similar to [13]. We deploy the sensors in a grid topology over a 100m X 100m area. We set the sensing range r of the sensors to 10m and the radio interference range to 50m. We did two simulations: one with network density of 1 node/r^2 and another with 2 nodes/r^2. Inter-sensor separation and the number of sensors deployed varies depending on the density. With 1 node/r^2 (respectively, 2 nodes/r^2), the inter-sensor separation is 10m (respectively, 7m) and the network size is 10x10 (respectively, 14x14). SS-TDMA [21] sets the TDMA period depending on the number of sensors falling in the interference range of a sensor. With 1 node/r^2 (respectively, 2 nodes/r^2), SS-TDMA sets the period to 50 (respectively, 100) slots, where one time slot = 30 ms.

We assume that the lifetime of a sensor is 20 minutes. We choose this value in order to ensure that the simulation completes within a reasonable time. (With density of 2 nodes/r^2, the simulation takes 3 days to complete. Typically, sensors are expected to work continuously for 1000 minutes. Simulating a sensor lifetime of 1000 minutes in TOSSIM, however, would approximately take 150 days to complete.) We simulate the lifetime of each sensor by maintaining a variable and decrementing it appropriately in each time slot.

In all our simulations, we set the timeout values for pCover as follows: X = 1 minute, Y = 2 TDMA slots, Z = 3 minutes, and S = W = 2 minutes. We randomly initialize the state of each sensor. We set OnThreshold and OffThreshold to 0.7 and 0.6. We consider that a network is "dead" when the *global coverage* of the network is less than a certain threshold even if all the alive nodes are working. Global coverage (or degree of coverage) is the percentage of the field that is covered by the working nodes. We define network lifetime as the duration from the beginning of deployment until the network is dead. We use 50% as the threshold in our simulations.

In our simulations, each link in the network has a bit error probability, representing the probability that a bit can be corrupted if it is sent along the link. Bit errors for each link is decided independently (using LossyBuilder, a Java program in TinyOS release) based on empirical loss data gathered from real world [25]. Next, we discuss our simulation results.

Coverage and network lifetime. In Figure 4, we show the degree of coverage and number of active sensors over time. In our simulations, we compute the global coverage for the entire 100m X 100m field and for the inner 80m X 80m field. The border sensors contribute only a part of their sensing range in the field and, hence, we consider the inner 80m X 80m field, where there is no such edge effect. As we can see from Figures 4(a) and 4(b), the sensors maintain the coverage at approximately the same level. With density = 2 nodes/r^2, initially (i.e., around 3 minutes), we observe a drop in the coverage. This is due to the fact that large number of sensors are initially set to active state (as a result of random initialization) and the number of active sensors fluctuate before converging to an appropriate number that maintains the coverage at a certain level (around 88.4%). Figure 4(c) shows the number of active sensors over time. As we can observe from the figure, this number remains at the same level until the point where the coverage starts dropping.

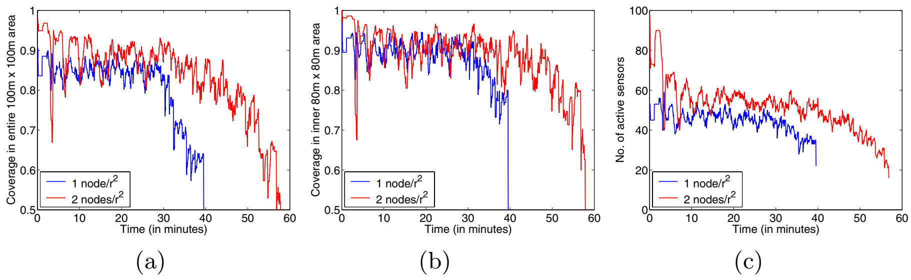
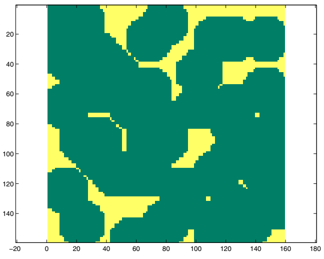


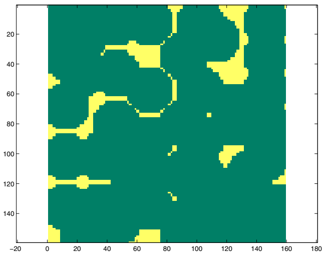
Fig. 4. Coverage and number of active sensors over time; (a) coverage of entire 100m X 100m area, (b) coverage of inner 80m X 80m area, and (c) number of active sensors

From Figures 4(a) and 4(b), we observe that the coverage is well maintained until one point, after which, the coverage drops suddenly, and the network dies in a short period. This shows that pCover maintains a balanced energy consumption as all sensors run out of power at around the same time. Also, we confirm the result in [13]; by sacrificing little coverage, the network lifetime is extended. Specifically, the lifetime with densities of 1 nodes/r^2 (respectively, 2 nodes/r^2) is around 39.55 minutes (respectively, 57.9 minutes).

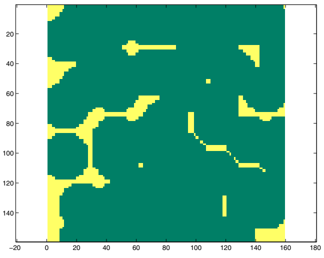
Quality of coverage. As mentioned in [13], in partially covered sensor networks, quality of coverage is an important metric. For example, in surveillance networks, it is measured in terms of how fast the sensors detect a target object. Since the sleep interval (i.e., X) is 1 minute, time to detect stationary objects in the sensor field is bounded by 1 minute. Additionally, since the sensors rotate their roles (working vs. sleeping), the set of active sensors changes continuously. Hence, an undetected "hole" is likely to be detected as the set of active sensors changes. In Figure 5, we show the snapshots of the field at different times.


(a) 10 minutes

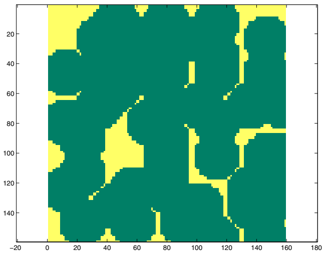
(coverage: 83.5%/87.6%)


(b) 20 minutes

(coverage: 89.9%/90.9%)


(c) 30 minutes

(coverage: 89.2%/91.6%)


(d) 40 minutes

(coverage: 85.2%/89.8%)

Fig. 5. Snapshot of the field with density = 2 nodes/r^2 (dark regions are covered). Coverage data below each subfigure shows the coverage of entire area and the coverage of inner 80m X 80m area respectively at that time.

From Figure 5, we observe that the location of "holes" change continuously. In surveillance networks, the intruder does not know the location of such holes. Hence, it is unlikely that the intruder can choose to move along the uncovered path. Therefore, the time to detect the intruder is small on average.

4 Lessons Learned in Prototyping Power Management Protocols

In this section, we discuss some of the lessons learned in prototyping power management protocols with ProSe.

Rapid prototyping and quick evaluation. Most of the power management protocols for sensor networks follow the event-driven model. For example, in pCover (cf. Section 3), we observe that a sensor switches to either working mode or sleeping mode whenever an event occurs (such as a timeout). Since guarded commands format is event-driven in nature, prototyping power management protocols with ProSe is straightforward. Furthermore, the time required to prototype protocols with ProSe is small. For example, the time required to prototype pCover with ProSe was in the order of few minutes. Also, the time required to specify LocalCoverage, a component used to compute the percentage of a sensor's sensing region covered by other active neighbors, was in the order of couple of hours. As mentioned in Section 3.2, this component uses only local data and, hence, we did not have to worry about communication. As a result, specifying this component in nesC/TinyOS platform was quick. By contrast, had we chosen to prototype pCover directly in nesC/TinyOS, we would have to deal with all low-level challenges of sensor networks and programming-level challenges of the platform. Based on our experience in developing protocols with nesC/TinyOS, we expect this effort to take considerable time (usually, few days to couple of weeks).

In short, ProSe provides a way to rapidly prototype power management protocols and generate the corresponding nesC/TinyOS implementation. Hence, the designers can quickly deploy and easily evaluate their protocols.

Preserving properties of interest. Since designers specify protocols in guarded commands format (with ProSe), they can analyze them for properties such as self-stabilization, fault-tolerance, and reliability. In addition, the designers can automatically add new properties to the guarded commands program. For example, the designers can use FTSyn [26] to automatically add fault-tolerance properties to their programs. If the transformation algorithm used to transform the input program (in shared-memory model or read/write model) into a model consistent with sensor networks (i.e., write all with collision model [16,17]) preserves properties of interest then ProSe also preserves such properties. ProSe implements the transformation algorithms proposed in [16,17] that preserve self-stabilization and fault-tolerance properties of the original programs. (We refer the reader to [15,16,17] for more details on how properties of interest are preserved in the transformed programs.) Thus, ProSe simplifies the design of power management protocols while ensuring that self-stabilization and fault-tolerance properties are preserved in the transformed programs.

5 Related Work

Work related to rapid prototyping of power management protocols can be categorized as: (i) programming platforms and (ii) power management protocols.

Programming platforms. Related work that deals with programming abstractions include [27,28,29,30] and tools for programming sensor networks include [31,32,20,33,34,35,36].

Programming abstractions. In [27], a state centric approach is proposed that captures algorithms such as sensor fusion, signal processing and control. In this model, the abstraction of *collaboration groups* hides the designer from issues such as communication protocols, event handling, etc. In [28, 29], *macroprogramming* primitives that abstract communication, data sharing and gathering operations are proposed. However, these primitives are application-specific (e.g., *abstract regions* for tracking and gathering [28] and *region streams* for aggregation [29]). And, in [30], *semantic services* programming model is proposed where users only specify the end goal on what semantic data to collect. Unlike [27,28,29,30], ProSe allows the designer to evaluate existing algorithms in the context of sensor networks. Moreover, since the programs are written in abstract models considered in distributed systems, ProSe permits the designer to verify the correctness of the programs as well as to manipulate the programs to meet new properties.

Programming tools. Techniques like virtual machine (e.g., *Maté* [31]), middleware (e.g., *EnviroTrack* [32]), library (e.g., *SNACK* [33]), and database (e.g., *TinyDB* [34]) are proposed for simplifying programming sensor network applications. However, these solutions are (i) application-specific, and/or (ii) restrict the designer to what is available in the virtual machine, middleware, library, or network. In [36], macroprogramming model, called *Kairos*, that hides the details of code-generation and instantiation, data management, and control is proposed. However, unlike [31, 32, 20, 33, 34, 35, 36], ProSe hides low-level details such as message collisions, corruption, sensor failures, etc. Moreover, ProSe does not require any runtime support.

Power management protocols. Related work on power management protocols for sensor networks include [7, 9, 8, 11, 10, 12, 13]. In [9], a sensor is allowed to go to sleep if and only if one of its neighbors can completely cover its sensing area. As identified by [7], this approach underestimates the coverage provided by neighboring sensors and, hence, it leads to energy waste. Additionally, both [7] and [9] require a global synchronization service. In [10], a coverage configuration protocol is proposed where a sensor can switch to sleep state if all *intersection* points inside its sensing range are at least k-covered (i.e., a point is covered by at least k sensors). However, unlike [13], this approach requires more number of active sensors.

Power management protocols proposed in [8,11,13] follow similar design principles. However, unlike [13, 11], in [8], a working sensor is awake continuously until its failure or depletion of power. In [8, 11], by controlling the range of messages transmitted, the density of working sensors is controlled. However, online estimation of transmission ranges and the number of working sensors are often difficult and inaccurate.

6 Conclusion

In this paper, we considered the problem of rapid prototyping of power management protocols for sensor networks. Since existing programming platforms (e.g., nesC/TinyOS) require the designers to be responsible for stack management, buffer management, and flow control, the designers of power management protocols prototype the protocols either by implementing a discrete event simulator or by modeling in a specialized simulator such as GloMoSim [14]. To enable rapid prototyping and quick evaluation of power management protocols in the target platform (e.g., nesC/TinyOS for Mica, XSM, or Telos based sensor networks), in this paper, we used ProSe, a programming tool for sensor networks. As a case study, we specified the power management program from [13] with ProSe, generated the corresponding nesC/TinyOS code, and evaluated its performance on TOSSIM [24]. We showed that the synthesized program provides partial (but high) coverage of the sensor field.

Since ProSe hides low-level challenges of sensor networks (e.g., message collision, corruption, synchronization, etc) and programming level challenges (e.g., buffer management, stack management, etc), the designers can rapidly prototype their protocols and generate code in the target platform. As a result, the development time and deployment time are small. In this paper, we illustrated this by prototyping pCover program [13]. We have also prototyped and evaluated the differentiated surveillance program [7] with ProSe (cf. [37]). Thus, with ProSe, we expect that the designers can rapidly prototype, quickly deploy and easily evaluate power management protocols in the target platform.

References

1. J. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6), 2002.
2. P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detection of rare, random, and ephemeral events. *In Proceedings of the Conference on Information Processing in Sensor Networks (IPSN)*, April 2005.
3. J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. *In Proceedings of the Fourth International Conference on Information Processing in Sensor Networks, SPOTS track*, 2005.
4. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *In Proceedings of Programming Language Design and Implementation*, 2003.
5. A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event driven programming is not the opposite of threaded programming. *In Proceedings of 2002 USENIX Annual Technical Conference*, June 2002.
6. O. Kasten and K. Römer. Beyond event handlers: Programming sensor networks with attributed state machines. *In Proceedings of the Fourth Internation Conference on Information Processing in Sensor Networks (IPSN)*, 2005.
7. T. Yan, T. He, and J. A. Stankovic. Differentiated surveillance for sensor networks. *In Proceedings of the First ACM Conference on Embedded Networked Sensing Systems (SenSys)*, November 2003.
8. F. Ye, G. Zhong, J. Cheng, S. W. Lu, and L. X. Zhang. PEAS: A robust energy conserving protocol for long-lived sensor networks. *In Proceedings of the International Conference on Distributed Computing Systems*, 2003.

9. D. Tian and N. D. Georganas. A node scheduling scheme for energy conservation in large wireless sensor networks. *Wireless Communications and Mobile Computing Journal*, May 2003.
10. X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. *In Proceedings of the Conference on Embedded Networked Sensing Systems*, 2003.
11. C. Gui and P. Mohapatra. Power conservation and quality of surveillance in target tracking sensor networks. *In Proceedings of the Tenth Annual International Conference on Mobile Computing and Networking*, 2004.
12. S. Ren, Q. Li, H. Wang, X. Chen, and X. Zhang. Analyzing object detection quality under probabilistic coverage in sensor networks. *In Proceedings of the International Workshop on Quality of Service (IWQoS)*, June 2005.
13. L. Wang and S. S. Kulkarni. Sacrificing a little coverage can substantially increase network lifetime. *In Proceedings of Third Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON)*, September 2006, to appear.
14. X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A library for parallel simulation of large scale wireless networks. *In Proceedings of the Workshop on Parallel and Distributed Simulations*, May 2002.
15. M. Arumugam and S. S. Kulkarni. Programming sensor networks made easy. Technical Report MSU-CSE-05-25, Department of Computer Science, Michigan State University, September 2005.
16. S. S. Kulkarni and M. Arumugam. Transformations for write-all-with-collision model. *Computer Communications (Elsevier)*, 29(2):183–199, January 2006.
17. T. Herman. Models of self-stabilization and sensor networks. *In Proceedings of the 5th International Workshop on Distributed Computing (IWDC)*, LNCS:2918:205–214, December 2003.
18. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, 1997.
19. M. G. Gouda and T. M. McGuire. Accelerated heartbeat protocols. *In Proceedings of the International Confernece on Distributed Computing Systems (ICDCS)*, 1998.
20. K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. *In Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services*, 2004.
21. S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing MAC for sensor networks. In S. Phoha, T. F. La Porta, and C. Griffin, editors, *Sensor Network Operations*. Wiley-IEEE Press, May 2006.
22. T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. *In Proceedings of the Workshop on Algorithmic Aspects of Wireless Sensor Networks*, 2004.
23. A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. *In Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, pages 221–235, 2001.
24. P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyOS applications. *In Proceedings of the Conference on Embedded Networed Sensor Systems*, 2003.
25. D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research, 2002.
26. S. S. Kulkarni and A. Ebnenasir. A framework for automatic synthesis of fault-tolerance. Technical Report MSU-CSE-03-16, Michigan State University, 2003.

27. J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing*, 2(4):50–62, 2003.
28. M. Welsh and G. Mainland. Programming sensor networks using abstract regions. *In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.
29. R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. *In Proceedings of the First Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.
30. K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, April 2005.
31. P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review*, 36(5):85–95, December 2002.
32. T. Abdelzaher et al. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. *In Proceedings of the International Conference on Distributed Computing Systems*, 2004.
33. B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). *In Proceedings of the Second ACM Conference on Embedded Networked Sensing Systems (SenSys)*, November 2004.
34. S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 2005.
35. R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. *In Proceedings of the International Conference on Information Processing in Sensor Networks*, 2005.
36. R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. *In Proceedings of the International Confernece on Distributed Computing in Sensor Systems (DCOSS)*, 2005.
37. M. Arumugam, L. Wang, and S. S. Kulkarni. Rapid prototyping of power management protocols for sensor networks: A case study. Technical Report MSU-CSE-06-26, Department of Computer Science, Michigan State University, July 2006.